Improving Fact Extraction of Framework-Based Software Systems

Jens Knodel
Fraunhofer Institute for Experimental Software Engineering (IESE),
Sauerwiesen 6, D-67661 Kaiserslautern, Germany
knodel@iese.fraunhofer.de

Martin Pinzger
Distributed Systems Group, Vienna University of Technology,
Argentinierstr. 8/184-1, A-1040 Wien, Austria
pinzger@infosys.tuwien.ac.at

Abstract

Modern software frameworks provide a set of common and prefabricated software artifacts that support engineers in developing large-scale software systems. Framework-related information can be implemented in source code, comments or configuration files, but in the latter two cases, current reverse engineering approaches miss important facts reducing the quality of subsequent analysis tasks. We introduce a generic fact extraction approach for framework-based systems by combining traditional parsing with lexical pattern matching to obtain framework-specific facts from all three sources. We evaluate our approach with an industrial software application that was built using the Avalon/Phoenix framework. In particular we give examples to point out the benefits of considering framework-related information and reflect experiences made during the case study.

Keywords: architecture recovery, fact extraction, frameworks, lexical pattern matching, parsing, reverse engineering

1. Introduction

Reverse engineering and in particular architecture recovery aim at extracting higher-level representations (i.e., the software architecture) from existing software systems and support engineers in assessing, maintaining, and evolving large-scale software systems. In order to produce such architectural views current reverse engineering tools process various artifacts available for the system under study such as source code, scenario profiles, documentation, domain information and expert knowledge [4].

Fact extraction from source code (i.e., finding pieces of information about the system) is a fundamental step of reverse engineering techniques and often has to be performed first [6], [10], [12], [14], [16]. That means before performing any high-level reverse engineering analyses or architecture recovery activities, available information in the source code has to be extracted and aggregated in a fact base. Such a fact base forms the foundation for further analysis tasks that are conducted next, either manually or (semi)-automatically using tools.

A common technique for extracting facts from source code is parsing. Basically, there exist several parsers for each programming language. However, for framework-based software systems fact extraction is more complex due to the reason that frameworks transcend the pure source code level with their own dialects and constructs. For instance, framework-specific statements may appear in source code comments, and configuration files are used to define certain properties of software systems. Typically, such information is removed by pre-processors or is ignored by parsers. The result is a reduced fact base lacking crucial information for further architectural analysis tasks.

In this work we introduce a generic fact extraction approach that allows the generation of a more usable and complete fact base for framework-based software systems. In addition to parsing that we use to extract the programming language-related facts from source code, we apply lexical pattern matching to extract the framework-related facts. Each framework-related statement is addressed by a pattern definition. The set of pattern definitions is stored in a pattern repository. From there a source code pattern matching tool retrieves all pattern definitions concerning a framework, investigates source code comments and configuration files, and produces facts about matched statements. The results of both extraction steps are then combined to an extended fact base that in contrast to other reverse engineering approaches also contains crucial framework-related information.

Our approach is open to various frameworks and only needs the adoption of the pattern definitions for the lexical analysis and the selection of an appropriate parser. By this means a more detailed fact base can be obtained at significantly lower costs than at adapting parsers to handle

Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)
1095-1350/03 $ 17.00 © 2003 IEEE

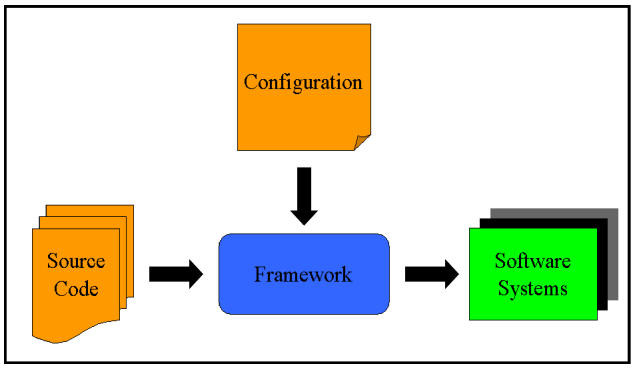

Figure 1: Frameworks

framework dialects. We will demonstrate this with an industrial software application that was built using the Phoenix/Avalon [17] framework.

The remainder of the paper is structured as follows: Section 2 provides the context for the case study by answering what are Java frameworks and why are they used. Our approach is described in detail in section 3. The case study itself with its results and experiences is described in section 4. Then section 5 provides an overview about current extraction techniques. Section 6 summarizes our work and draws conclusions.

2. Frameworks

Frameworks provide a set of common and prefabricated pieces of software that developers can use, extend or customize to build software applications. In its simplest form a framework is just a body of tried and tested code as stated by Sheil in [18] that, however, reduces development time, improves quality and enhances maintainability. In [24], Valerio states that frameworks provide reusable components, which implement functionality and allow tailoring and customizing the application to the customer needs.

As depicted by Figure 1, frameworks add generic functionality to the product-specific source code. Based on the configuration of the software system, the developers can easily create several slightly different systems just by adopting some entries within the configuration. This kind of variation mechanism can be used to address customer-specific requirements.

Embedding software applications into a framework often can be implemented in different ways, basically, including source code, but also comments and configuration files.

Phoenix/Avalon [17] is an example for a framework that is used to build large distributed software applications in Java.

```
/**
* Removes the application from
 * container
*
* @phoenix:mx-operation
*
* @param name the name of
*         the application
*/
void removeApp( String name )
{
  ...
}
```

Figure 2: Example of an mx-operation doclet tag

2.1. The Phoenix/Avalon Framework

The Avalon project (see [2]) is an effort to create, design, develop and maintain a common framework and set of components for applications written using the Java programming language. It allows components of varying scale to be managed via a specific set of lifecycle methods. Complete applications may be managed in a server oriented container such as Phoenix.

Phoenix is a micro-kernel designed and implemented on top of the Avalon framework. It provides a number of facilities to manage the environment of server applications. Such facilities include log management, class loading, thread handling, security, and the java management extensions (JMX).

Via JMX, it is possible to control and to manipulate an instance of Phoenix at run-time. Such an instance is composed of variables, components, applications and blocks. The JMX functionality is generated automatically during build-time by doclet tags. These tags are written as Javadoc comments (with a "@phoenix:" prefix) directly into the source code files. There are 4 major doclet tags:

- mx-topic: the mx-topic tag marks a class or interface as eligible for management (i.e., it can be accessed by JMX).
- mx-attribute: attributes can be read or written by getter or setter methods, if they have the mx-attributes tag.
- mx-operation: these are methods, which can be invoked via JMX.
- mx-proxy: The mx-proxy tag is used to indicate that a proxy class should be used to manage some aspect(s) of an object. At runtime, the management system will create an instance of the proxy class passing in a reference to the managed object in the constructor. Management calls are then made on the proxy instead of the managed

object.

Developers do not need to implement these tags by hand, instead they just use the doclet tags. And how this functionality gets used is up to the Phoenix/Avalon framework. User interfaces are generated automatically for the management of the interaction of the framework and its application.

Another way of gaining benefits when using a framework are complex, mostly XML-based configuration files. For example, Phoenix/Avalon facilitates the configuration of:

- Initial values for variables or class attributes.
- Default values for parameters in method signatures.
- Communication mechanisms (e.g., threads, sockets).
- Communication chains between components within the framework (e.g., which components provide data, which consume them)
- Data sources (e.g., databases, files)
- Component dependencies (which components have be activated for performing a task)

Since the configuration of a framework defines numerous properties of the software system, it is important that the fact extraction process covers this kind of information, too.

3. Framework-Based Fact Extraction

Basically, we combined the conventional parsing process with lexical pattern matching to extract the framework-related information from various information sources such as source code, comments, and configuration files. Figure 3 shows our generic extraction process in IDEFØ [9] (Integrated Definition Language) depicting the basic input sources processed and results produced by our process. A detailed description of the five process steps is given in the following subsections.

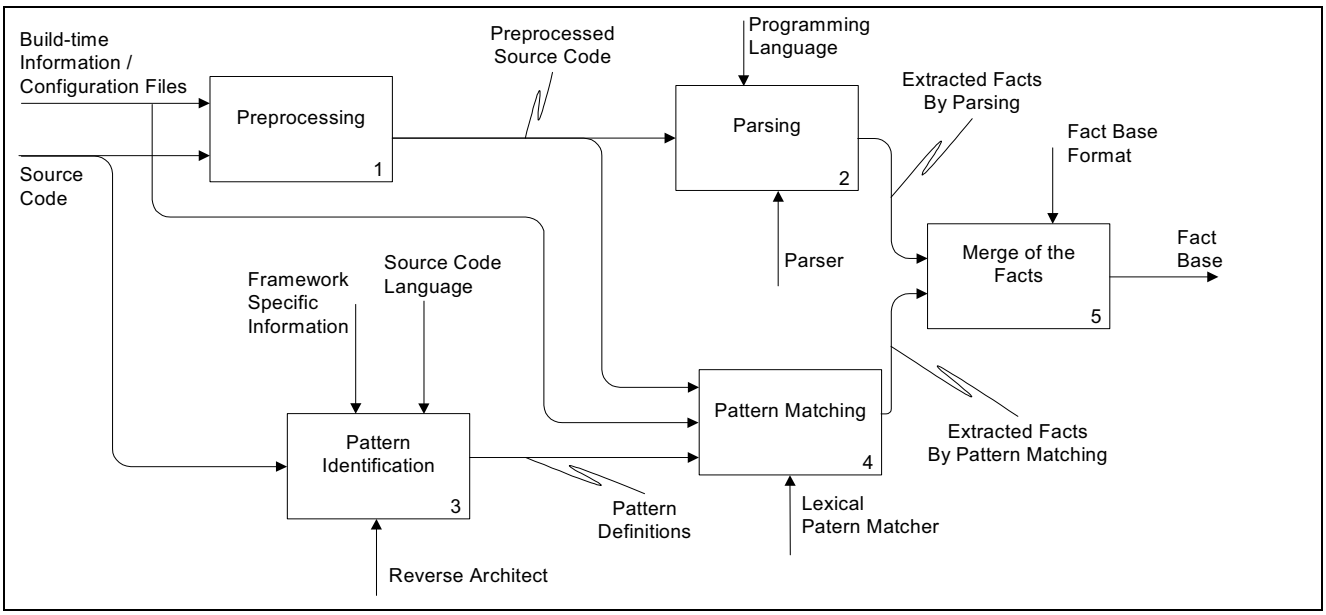

Figure 3: The fact extraction approach

3.1. Step 1: Preprocessing

The source code of a software system often contains preprocessor commands. These commands offer the possibility to work with symbolic constants, macros and conditions. Conditions may introduce certain variants of the same software system varying in their behavior. This means that the resulting fact bases for two variants may be different, too. Therefore the approach analyzes only one concrete instance of the source code, a preprocessed one. The preprocessing step uses build-time information to find out about macro names and conditions that create a concrete instance. The parser and the lexical pattern matcher will work only with the preprocessed code.

3.2. Step 2: Parsing

A parser is a program that receives input in the form of source code instructions and breaks them up into parts (for example, objects, methods, and attributes). This collected data is filled into the fact base. Furthermore, dependencies among the extracted entities are added to the fact base. Depending on the source code language, an appropriate parser is chosen. Typically, each programming language has its own grammar that specifies the syntax. For this reason, there is no all-purpose parser.

3.3. Step 3: Pattern identification

In order to address the framework-specific properties of the software system, the reverse architect specifies the criteria for the parts of the source code ignored by the parser. The reverse architect has to turn his attention to the unambiguous definition of each pattern. Otherwise the results might get falsified. So-called false positives are matches that fulfill the specified criterion, but they rather should be left out.

The same prerequisites count for the definition of the patterns addressing information defined in configuration files. The reverse architect first has to identify the parts of the framework configuration, which are of importance to the system, and then he has to figure out the characteristics of the found location. After that he can formulate the specification of the pattern. All pattern definitions are stored in a pattern repository for two reasons:

- To allow the composition of complex patterns based on less complex or trivial ones.
- To reuse already available pattern definitions in the context of other frameworks or other systems based on the same frameworks.

The reverse architect can reuse pattern definitions for other software systems operating with the same framework. The reason for this is that the mechanisms the framework provides will not change when embedding different applications into the same framework. Furthermore, it may be possible to apply pattern definitions when analyzing a system embedded into another framework. For instance, the configuration files of different frameworks can all be XML-based and use the same keywords to define software systems properties.

3.4. Step 4: Pattern matching

Having the pattern descriptions, the analysis with the lexical pattern matcher is performed in two cycles. The first round addresses the preprocessed source code, while the second one takes care of the configuration. The extracted facts complete the data already contained in the fact base.

Depending on the chosen lexical pattern matcher, this step can be performed automatically by the tool. Inputs to this step are on the one hand the pattern definitions, and on the other hand the artifacts of the software system, the source code and the build time information (i.e., configurations).

3.5. Step 5: Merge of the facts

After having the results of parsing and of pattern matching, we have to merge both results in the final step of our approach. For further analysis steps both results have to be in the same format in the fact base as well as the same syntax for the entries in the fact base has to be used. For instance, if a method is extracted in the format <class_name>.<method_name> by the parser then the pattern matcher should produce the same format (i.e., use the dot as class method delimiter). This conceptual merge ensures that the source code facts extracted by one of the techniques match each other in the fact base.

In the end, the resulting fact base consists of information from different information sources but in the same format. It can now serve as a basis for further reverse engineering activities like architecture recovery.

4. Case Study

The case study dealt with an industrial software system programmed in Java. The system is embedded in the Phoenix/Avalon framework, and it can be easily adjusted to new requirements via XML-based configuration files. About 20 packages containing nearly 200 classes result in approximately 30K lines of code. Since no preprocessing

commands were used in the source code, the first step of the approach was left out.

We used the Rigi tool [23] through out the whole case study to visualize the results we gained. For this reason, we decided to build up the fact base as one single file in the Rigi Standard Format (RSF). Some small Perl scripts took over the conversion of the collected data into the RSF format, as well as the merging of the different information sources within the fact base.

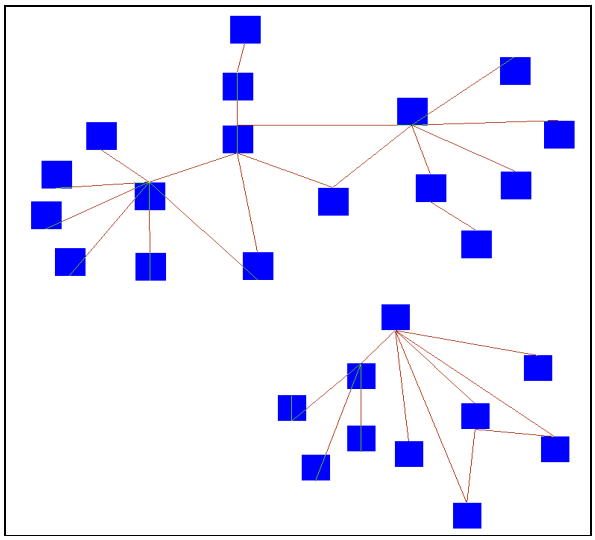

Figure 4: Subset of the call graph

4.1. Parsing

We used javagen, a tool based on GEN++ (see [5]) to extract the facts directly from the source code files provided by the industrial partner. The parser extracted the following facts and relations:

- Classes: the application-specific classes
- Methods: the methods of the classes
- Attributes: these are class members, method variables and static variables
- Inheritance information: the parent-child relation for each class
- Method invocation: which method is invoked by whom
- Attribute access information: which methods access which attributes

When reviewing the results generated by the javagen parser (we applied manual code inspections as described in [6]), we discovered that parsing alone was not sufficient. We had to take the configuration files into account as well. For instance, the call graph was relatively sparse and disjointed. Additionally, it showed some call sequences, where the code entities were only connected among themselves. The source code and the configuration then exposed that the software system consisted of several components, each running in its own thread. The communication took place via queues, and the parser was unable to extract this information (i.e., the providers and consumers of the different queues).

```
<queuedefaults>

<processSize>500</processSize>
  </queuedefaults>

  <queue name="q1">
      <maxSize>250</maxSize>
      <processSize>5</processSize>
      <sink name="name1"
            block="BLOCK_A"/>
      <source name="name1"
            block="BLOCK_B"/>
  </queue>

  <queue name="q2">
              ...
  </queue>
```

Figure 5: Example of a configuration file

Figure 4 shows another example of a parsing problem. A subset of the call graph is visualized with the Rigi tool, extracted by the javagen parser. The boxes stand for methods, and a line between two boxes means one methods calls the other. The upper and the lower part of the graph are not connected. Reviewing the code showed that the two parts are related to each other. But the parser missed this information. We then analyzed the specific parts directly in the source code, in order to learn about the reasons why the parser missed those parts. The parser could not extract some calls to methods of a "private final" class instance, if the method was called with a preceding "this" (i.e., this.<class_instance>.<method>). We assume the reason for this is a bug in the parser.

Furthermore, the parser was not able to detect that certain source code entities are related to JMX functionality. The relations to JMX are part of source code comments, which are not analyzed by the parser. However, since the application can be controlled and manipulated directly by the user via JMX, it is crucial information, which might have impact on further architectural analysis activities.

4.2. Pattern identification

Pattern identification occurred in two places, the source code and configuration files. Concerning source

code we were interested in Phoenix-related JMX doclet tags that are contained in source code comments. In particular we focused on the tags “mx-topic”, “mx-proxy”, “mx-operation” and “mx-attribute”. For each such doclet tag we can specify a corresponding source code pattern to match it and output the framework-related information accordingly.

Configuration files can contain a lot of information. The software system holds several coarse-grained components, which share data over several queues. The configuration decides about how many queues are present in the system, and which components operate as provider and/or consumer to a specific queue. Since these connectors characterize one of the important properties of the system that is not extractable by a parser with reasonable effort, we focused on the detection of these queue connections.

Figure 5 shows an excerpt of such a configuration file. Each queue is indicated with the keyword “<queue>”, the connected components are indicated by the keyword “<block>”. If “<block>” follows a “sink”, it provides data to the queue, otherwise it consumes entries out of the queue.

The base classes of each component, in the context of the Avalon framework also called block, could be extracted out of another part of the configuration. In this way the queue could be mapped to certain source code entities, and thus, the fact base was accumulated with this type of information.

All pattern definitions were centered within a pattern repository. This allows us to reuse the definitions in future analysis activities.

4.3. Pattern matching

For the lexical pattern matching we used Revealer [18], which is a lightweight source model extraction tool that combines lexical with syntactic analysis capabilities. It is based on regular expressions and provides basic pattern elements that can be combined to simple and complex (i.e., hierarchical and nested) pattern definitions in a tree-like structure.

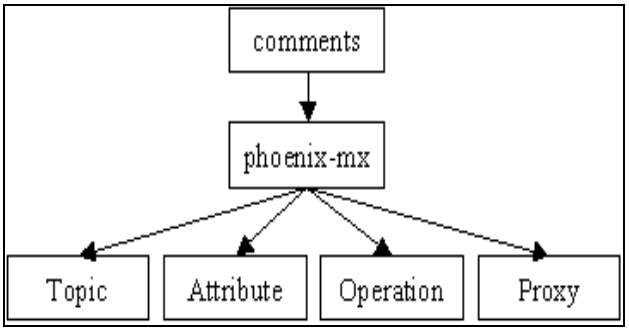

Figure 6: Structure of Phoenix-related patterns

Besides the combination of pattern elements Revealer pattern definitions support reuse of existing pattern definitions and the specification of match and output actions per pattern element. Basically, match actions are used to further investigate the text of a matched pattern element and output actions are used to output match results in a proper data format such as for example RSF.

In the context of our case study we developed a set of pattern definitions to extract Phoenix-related statements of interest for our analysis tasks. We organized the pattern definitions in a hierarchical way so that we were able to increase reuse of pattern definitions and control the extraction process. Figure 6 shows the basic structure of our pattern definitions to match Phoenix JMX statements.

The matching process starts with the “comments” pattern definition to match Java multi-line comments and subsequent class or method signatures. Matched pattern instances are sent to the “phoenix-mx” definition. This pattern definition filters out non-Phoenix-related comments and sends the Phoenix relevant comments to a number of pattern definitions that are used to match the Phoenix-related statements. Latter pattern definitions output the match results about JMX controlled topics, attributes, operations, and proxy objects in RSF format.

Figure 7 depicts the details about the Revealer pattern definition we applied for matching Phoenix-related JMX doclet tags. The pattern definition consists of five sections each indicated by an XML comment:

1. Specifies basic pattern elements to match the “@phoenix:” keyword that indicates the comment as a Phoenix-related one containing JMX doclet tags. Furthermore, it specifies a pattern element to match the source code line succeeding the comment that is either a class or a method signature.
2. Links the single pattern elements specified before to the root pattern definition.
3. References existing pattern definitions to be reused. These definitions match the JMX doclet tags of interest.
4. Specifies that each matched instance of the root pattern definition is sent to the pattern definitions specified in the reuse section.
5. Organizes the links that integrate the reused pattern definitions into the root pattern definition.

For the investigation of the configuration files we proceeded in the same way but with different pattern definitions to extract queues related classes. We explored the configuration file in order to describe the pattern in an unambiguous way.

Basically, each class is running in a different thread, the data flow between the classes happens via queues. A class can consume data out of a queue, or it can provide

data for a queue. Every consumer checks regularly if there is some new data to process in the queue. The data producer and the data consumer classes were specified in the configuration file of the software system. The developer is able to change the data flow of the system just by changing the correspondent entries in the configuration file.

Figure 8 shows the results of the configuration file analysis, a communication chain operating with queues. The box in the top depicts the first data producer class that receives its input data from an external information source. It handles the incoming information and dispatches the data to three other classes via a queue. Then the data is processed. For instance, database entries are adjusted, or the inputs are announced to the user.

By analyzing the configuration files of the software system, the communication chain of the queues was captured. By only working with the source code alone, this important information about the software system would have been lost.

Figure 9 shows the same subset of the call graph as Figure 4, but this time including both, the extracted facts by parsing and by lexical pattern matching. In this case, there were two more connections extracted between the two subsets by lexical pattern matching. The combination of parsing and lexical pattern matching carried out that the call graph now is more complete.

```
<pattern root="main" dirs="." files=".*\.java">

     <!-- 1. Phoenix management extension pattern -->
     <pe id="phoenix-mx" type="Definition"/>
     <pe id="phoenixSeq" type="Sequence"/>
     <pe id="phoenix" type="StringExp">
           <attr name="string">@phoenix:</attr>
     </pe>
     <pe id="phoenixDecl" type="RegExp">
           <attr name="regexp">[^\n]+</attr>
     </pe>

     <!-- 2. Relations between pattern elements -->
     <rel from="main" to="phoenix-mx" type="contain"/>
     <rel from="phoenix-mx" to="phoenixSeq" type="contain"/>
     <rel from="phoenixSeq" to="phoenix" type="contain"/>
     <rel from="phoenix" to="phoenixDecl" type="next" start="0"/>

     <!-- 3. Reused pattern definitions -->
     <pe id="mx-topic" type="Definition"
                        reuse="mx-topic.xml#phoenixTopic"/>
     <pe id="mx-proxy" type="Definition"
                        reuse="mx-proxy.xml#phoenixProxy"/>
     <pe id="mx-operation" type="Definition"
                        reuse="mx-operation.xml#phoenixOperation"/>
     <pe id="mx-attribute" type="Definition"
                        reuse="mx-attribute.xml#phoenixAttribute"/>

     <!-- 4. Investigate phoenix declaration in more detail -->
     <pe id="innerPhoenixDecl" type="SendTo"/>

     <!-- 5. Link pattern definitions to phoenixDecl element -->
     <rel from="phoenixDecl" to="innerPhoenixDecl" type="sendTo"/>
     <rel from="innerPhoenixDecl" to="mx-topic" type="contain"/>
     <rel from="mx-topic" to="mx-proxy" type="next"/>
     <rel from="mx-proxy" to="mx-operation" type="next"/>
     <rel from="mx-operation" to="mx-attribute" type="next"/>
</pattern>
```

Figure 7: Revealer pattern definition to match Phoenix-related JMX doclet tags

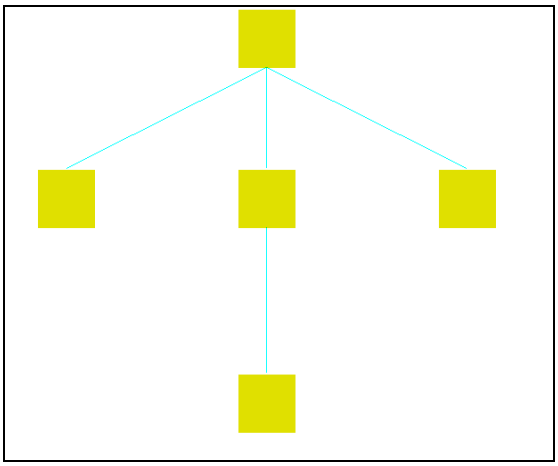

Figure 8:Classes connected via queues

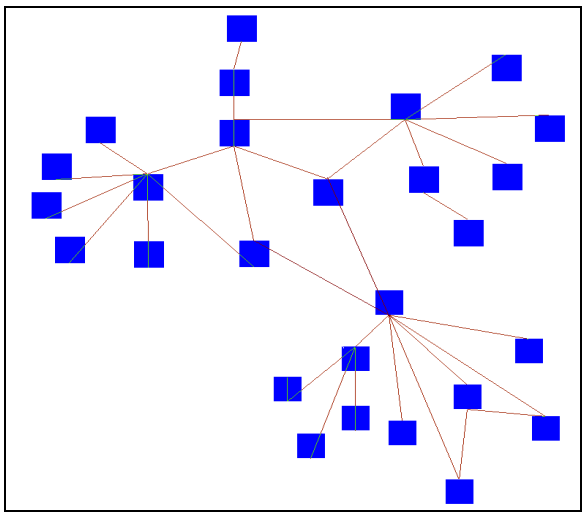

Figure 9:Call graph subset after merging results

4.4. Wrap up

To summarize the case study, we extracted facts out of a framework-based software system to prepare further analyses. We recognized that we missed crucial information by just using a standard parser. For this reason, we combined the parser with a lexical pattern matcher in order to minimize the effort spent on fact extraction. The specification of one single pattern took averaged not more than 1 hour. This time includes the isolation of a pattern out of the framework, the comprehension of it as well as the writing down of the description. The most part was spent to find out about what elements formed the pattern. But, compared to a parser enhancement, where the implementation of the enhancements would cost a lot, the lexical analysis was the more effective alternative in our case [20]. A problem we encountered in the beginning was that we had different formats for the extracted facts. This was easily resolved by adapting the Perl scripts when merging the facts into one common fact base. It is necessary that the extractors produce the same format for the fact base (i.e., identifiers of source code entities must have the same format).

Table 1: Doclet tags

Doclet Tag	Number of Occurrences
mx-topic	20
mx-attribute	22
mx-operation	10
mx-proxy	2

Table 1 gives information about the numbers of doclet tags that have been matched using the lexical pattern matcher. Furthermore, we were able to detect all of the specified queues responsible for the main data flow of the software system.

The lexical analysis with the Revealer tool was able to complement the results already gained by parsing. For instance, the disjointed call graph was noticeable more connected in the end. The fact base was filled up with data fundamental for further architecture recovery activities. Thus, we claim that architecture recovery of framework-based software systems benefits from the introduced generic extraction approach.

5. Related work

There are a variety of techniques and tools for the extraction of facts about a software system. On the one hand there are a lot of parsers that exists for the different source code languages.

To parse Java, for instance, Korn et al. developed the Chava parser as described in [11] as part of the CIAO toolkit. Chava extracts information from Java code about classes, methods, fields and their relationships into a relational database. The database can be queried to learn about the software system. In [3] Bowman et al. compare different techniques to extract information from Java software systems, namely parsing, disassembling and profiling. The proposed extraction techniques by Korn or Bowman differ from our approach because they are not able to capture the framework-related facts. The fact base in their approach will miss some important information about the software system. There are a lot of other parsers available, but they also have the problem that they can handle only the standard language constructs and not the framework-related properties of the software system under investigation.

Lexical analysis can offer a solution to certain extraction problems and reduce costs of generating a more usable fact base. There are several lexical-based analysis tools available that could be used for extraction of

framework-related information. Well known examples are lex [13], awk [1], or Murphy's Lexical Source Model Extraction (LSME) approach [15]. Lex and awk define patterns as rules consisting of a regular expression and an action. Matches are stored in specific variables that can be accessed by actions to generate proper output data. However, both tools lack of capability to express syntactic constructs as used by programming languages and framework dialects. LSME overcomes this problem and allows for the specification of hierarchical related expressions. One drawback of LSME, however, is that it provides only two classes of tokens (single-character and identifier) for specifying pattern definitions, which lowers expressiveness.

Concerning fact extraction of framework-based software systems Pinzger et al. in [19] introduced an iterative and interactive extraction process that combines component inspection techniques with source code analysis to extract higher-level representations of COM+-based software applications. Whereas their approach is focused only on component frameworks and in particular on Microsoft's COM/COM+ our approach is more generic and considers arbitrary frameworks.

Succi et al. describe in [22] an approach to extract frameworks with the help of domain analysis. The main difference between our and their approach is main goal. We are building a more complete fact base, while they are trying to build a reusable framework-based on existing systems. We address with our approach already existing third-party frameworks rather than the extraction of a framework comprising reusable components.

6. Conclusion

Software development organizations apply frameworks more and more to benefit from the advantages they offer. Modern framework functionality and mechanisms are often implemented in source code comments and configuration files. Parsing usually misses such information, so there is a need for action in reverse engineering to address the framework-specific differences. Instead of enhancing the implementation of the parser we chose lexical pattern matching to solve the framework-related problems in the extraction process. The specification of the patterns used in the software system has cost less effort than a possible enhancement of the parser.

In this work we introduced a generic approach for the fact extraction of framework-based software systems, and we showed the usefulness of our approach with an industrial case study. Further reverse engineering or architecture recovery steps will benefit from the more complete foundation gained through the combination of parsing and lexical analysis.

Our generic fact extraction approach is open to any parsers, any lexical pattern matcher and any framework. This means as soon as there is a parser for a specific source code language and we can specify pattern definitions for the framework-related properties, our approach will build a more complete fact base for the software system under investigation. Both, results of parser and lexical pattern matcher will be merged into one common fact base, building the basis for further architectural analysis.

Ongoing work will perform further case studies, where other framework-based systems will be analyzed, and the pattern definition repository will be extended through this work.

In future, we will extend our catalogue of patterns, and when the reverse architect is analyzing a new framework-based software system, he can apply the already existing patterns. We expect then savings in time and effort for the pattern definition due to pattern reuse.

7. Acknowledgements

This work is partially funded by the European Commission under EUREKA 2023/ITEA-ip00004 'from Concept to Application in system-Family Engineering (CAFÉ)'.

8. References

[1] A. V. Aho, B. W. Kernighan, and P. Weinberger, "Awk - a pattern scanning and processing language", Software Practice and Experience, 9(4):267-280, 1979

[2] The Apache Avalon Project, http://avalon.apache.org/

[3] I. T. Bowman, M. W. Godfrey, R. C. Holt, "Extracting Source Models from Java Programs: Parse, Disassemble, or Profile?", http://plg.uwaterloo.ca/~itbowman/papers/javasrcmodel.html

[4] E.J. Chikofsky, J.H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy", IEEE Software, January 1990, pp. 13-17

[5] P. Devanbu,, "A language and front-end independent source code analyzer", Proceedings of ICSE 1992, Melbourne, Australia

[6] M. E. Fagan, "Design and code inspections to reduce errors in program development", IBM Systems Journal, vol. 15, no. 3, pp. 182-211, 1976

[7] G.Y. Guo, J.M. Atlee, R. Kazman,, "A Software Architecture Reconstruction Method", Proceedings of the 1st IFIP Working Conference on Software Architecture, pages 15-33, San Antonio, Texas, USA, February 1999

[8] IEEE Std 1471-2000, “IEEE Recommended Practice for Architectural Description of Software-Intensive Systems”, IEEE-SA Standards Board, September 2000

[9] Integrated Definition Language, http://www.idef.com/idef0.html

[10] R. Kazman, S.J. Carriere, ”View Extraction and View Fusion in Architectural Understanding,”, Proceedings of the Fifth International Conference on Software Reuse, 1998

[11] J. Korn, Y-F. Chen, E. Koutsofios, “Chava: Reverse engineering and tracking of java applets”, Proceedings of the 6th Working Conference on Reverse Engineering, WCRE 1999, pages 314-325

[12] R. L. Krikhaar,, Reverse Architecting for Complex Systems, Proceedings of the International Conference on Software Maintenance, ICSM 1997

[13] M. E. Lesk,, “Lex - a lexical analyzer generator”, Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, New Jersey, October, 1975.

[14] A. von Mayrhauser, J. Wang, Q. Li, “Experience with a Reverse Architecture Approach to Increase Understanding,”, IEEE International Conference on Software Maintenance (ICSM), 1999

[15] G. C. Murphy and D. Notkin, “Lightweight lexical source model extraction”, ACM Transactions on Software Engineering and Methodology, 5(3):262-292, July 1996.

[16] L. O’Brien, “Architecture Reconstruction to Support a Product Line Effort”, Software Engineering Institue, Technical Note CMU/SEI-2001-TN-015, July 2001

[17] The Phoenix Project, http://avalon.apache.org/phoenix/index.html

[18] M. Pinzger, M. Fischer, H. Gall, and M. Jazayeri, “Revealer: A Lexical Pattern Matcher for Architecture Recovery“, Working Conference on Reverse Engineering, pages 170-178, October 2002

[19] M. Pinzger, J. Oberleitner, and H. Gall, “Analyzing and Understanding Architectural Characteristics of COM+ Components”, Proceedings of the 11th International Workshop on Program Comprehension, May 2003, page 245-250

[20] H. Reubenstein, R. Piazza, and S. Roberts, “Separating parsing and analysis in reverse engineering tools”, Proceedings of the Working Conference on Reverse Engineering, pages 117-125, 1993

[21] H. Sheil, “Frameworks save the day”, JavaWorld, 2000, September 29, http://www.javaworld.com/javaworld/jw-09-2000/jw-0929-ejbframe_p.html

[22] G. Succi, A. Valerio, T. Vernaza, M. Fenaroli, P. Predonzani, “Framework extraction with domain analysis”, ACM Computing Surveys 32,

[23] S.R. Tilley, K. Wong, M.-A.D. Storey, H.A. Müller, “Programmable reverse engineering”, International Journal of Software Engineering and Knowledge Engineering, pages 501-520, December 1994

[24] A. Valerio, “Special issue on the effects of frameworks and patterns on software reuse”, ACM SIGAPP Applied Computing Review, September 1997, Volume 5, Issue 2